ED-cPSD Documentation

Andre Adam, Guang Yang, Huazhen Fang, Xianglin Li

Last Updated: April 9, 2025

Contents

Introduction

This document serves as a reference and documentation for ED-cPSD, a software based on a novel algorithm of sequential erosion-dilation for continuous phase-size distribution (ED-cPSD) estimation. This document is organized into a few different chapters which serve as a reference for both users and contributors.

Chapter 1 is dedicated to helping new users get the software running and generating meaningful results immediately. Chapter 1 will cover the installation process, basic GUI usage, running code decoupled from the GUI, preparing inputs with ImageJ [1] and/or simple Python scripts (OpenCV), expected outputs, and interpreting outputs.

Chapter 2 presents an in-depth mathematical description of the algorithm working behind the scenes to calculate the continuous phase-size distribution (cPSD). This chapter also includes implementation details for both the standalone code and the GUI computational model. These implementation details are important for researchers looking to contribute to and/or fork this project.

Chapter 3 presents several examples and case studies of the use of this software/algorithm. Each case study highlights quintessential aspects of the algorithm, such as interpreting the output data given different inputs, comparisons with other cPSD and discrete phase-size distribution (dPSD) algorithms, and use cases in materials science and energy research. As more research involving this algorithm is published, the case-study section will grow accordingly.

Finally, chapter 4 provides additional information such as acknowledgments, code licensing information, and references.

Chapter 1

Quick-Start Guide

This chapter will be a brief introduction on how to download the GUI and produce results with minimal setup required. This chapter will also focus mainly on Windows users. Some special instructions for Linux users are included in a separate section.

1.1 Setup

1.1.1 Method 1: Download Executable

The first step to get the software running is to download it directly from one of the source repositories. For users who are inexperienced with Git, it is recommended to go directly to the project GitHub repository at `https://github.com/adama-wzr/ED_cPSD`. From there, select the green "Code" button, and select the option "Download Zip", as shown in figure 1.1.

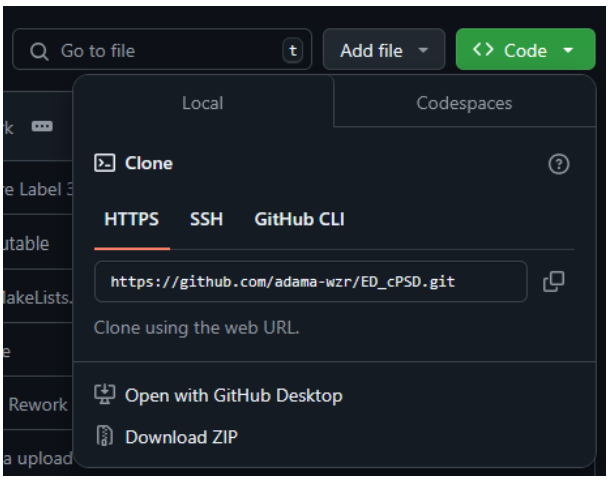

Figure 1.1: Select the green button, and then select the option to download the zip containing all files.

Once the zip is downloaded, it can be extracted to the desired location on the computer. The executable can be found in the folder "ED_cPSD/build/Desktop_Qt_6_8_1_MinGW_64_bit/". The executable is titled "ED_cPSD.exe". You can right click the ".exe" and create a shortcut, then copy the shortcut to a place where it is more accessible, or just search for the executable using Windows search.

Verify at this stage that you are able to open the software. If you can open it, then everything should run. Skip to one of the examples to learn how to use the software.

1.1.2 Method 2: Build Project

Statically linking libraries is relatively memory consuming on a computer, and might not be the best way to build the software for the experienced user. Thus, some may instead choose to build the executable.

In order to do that, make sure that the following requirements are met:

- Download the "GUI_src" folder from GitHub.
- Ensure a modern C++ version is installed.
- Ensure you have Qt6.8 installed and it is in the PATH (any recent major release of Qt should work, but only 6.8 was tested).
- (optional) Qt Creator 15.0 or newer (other versions might work).

Next, in the Qt Creator, select the option to "Open Project", navigate to the folder where "GUI_src" was saved, and select the "CMakeLists.txt" file. A window will then appear asking the user to configure the project, but no additional configurations are need. Simply click the "Configure Project" button on the right side of this window, and Qt will build it for you.

At this stage, the GUI should be ready to run within the Qt Creator. Simply press "Run" or use the shortcut "Ctrl-R". The software can also be run outside of Qt Creator if Qt is included in the PATH environment variables. A easier alternative is to use the Qt Deployment Tool for Windows `https://doc.qt.io/qt-6/windows-deployment.html`, which will statically link the necessary Qt libraries directly to the executable file's folder.

1.2 Examples

In the code repository, you will find a folder labeled "Examples", with three simulation examples (one in 2D and two in 3D). This folder includes the input files and expected outputs. But in this section, we will examine how to reproduce those examples.

Make sure the source files containing the structure information (`.csv` and `.jpg`) are saved at your computer. The input files and simulations will be run using the GUI, which can be installed anywhere on your computer.

1.2.1 2D Example

This example will demonstrate how to process an image using ImageJ and properly format it to be an acceptable input to the code. The image in question is `cRec2000.jpg`, a 2000×2000 reconstruction of a carbon electrode, featured in Adam 2022 [2].

The first step is to open the image using ImageJ. If the image is not of `.jpg`, the **FIRST** step is to go to the "File" tab, select "Save As", and then save as `.jpg`.

Once the image is saved as `.jpg`, go to the "Image" tab, select the option "Type", and make sure the image is saved as "8-bit". If there is a need for thresholding, go to the "Image" tab, "Adjust", and then "Threshold". Here the threshold can be adjusted. The software always assumes that white is pores, and black is particles. The image can either be converted to binary at this stage, or the thresholding value to separate particles from pores can be entered into ED-cPSD directly. Remember to save the image after making these modifications.

Once the image is saved, you can verify the changes by right clicking on the image, selecting "Properties", then "Details" to ensure that "bit-depth" is 8-bit. That indicates a single-channel grayscale image. An RGB image will usually say 24 or 32-bit, which is not desired.

Next, generate the input file for the simulation. Open the ED_cPSD software, and navigate to the 2D tab if necessary. Under the 2D tab, the first field asks for the image name, which in this example is "cRec2000". Do not include the file extension, `.jpg`, as that is automatically entered.

Below the file name field, the threshold can be adjusted on a scale from 1 to 255. You can also select the desired outputs. For this example, select all options. Leave the output file names empty to use the default names.

The remaining options relate to simulation parameters. The number of threads simply controls the number of CPU threads OpenMP will use for the simulation. The radius offset, used for smoothing, means the simulation will start at a radius of $r = r_{\text{off}}$ instead of $r = 1$. The "Max. Radius" option sets a maximum radius cutoff for the simulations. ED-cPSD will not scan beyond $r = r_{\max}$ even if the solution is incomplete. Finally, the "Verbose" option controls whether output is printed to the window (GUI) or command line (for running the code on the command line).

Press the "Generate" button to display the input file on the screen, as shown in figure 1.2.

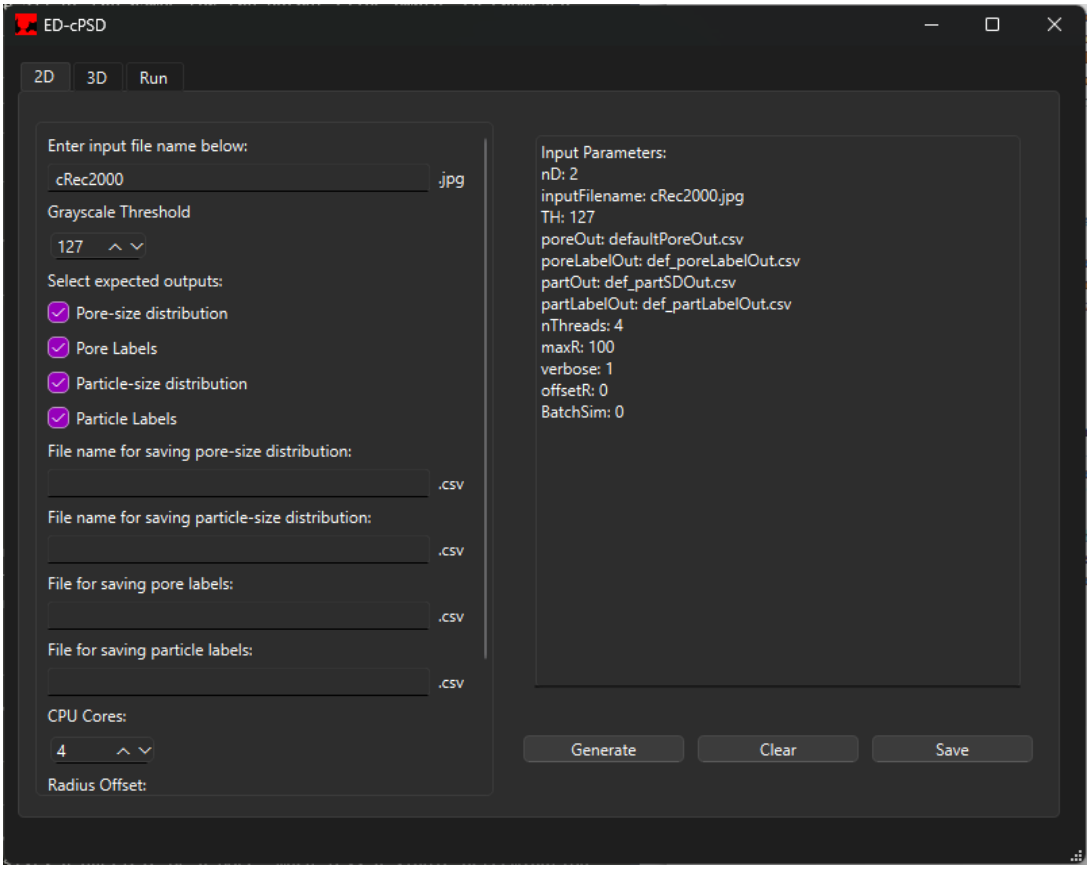

Figure 1.2: Example input file on ED-cPSD for the 2D example.

Press the "save" button, and be sure to save the input in the same folder as the image you are simulating. Then, go to the "Run" tab, select the same folder again or manually enter the path to the folder. At this stage, the simulation is ready to run. Simply press the "Run" button and watch the results being printed to the text tab on the right-hand side of the screen.

Even though the image is 2000×2000, both pore and particle size distributions will be obtained within a few seconds. Note that the GUI adds a 0.25 *ms* delay every time it prints something to prevent pointers from being lost to garbage collection. This may cause the output to take longer to print than the actual simulation. If this is an issue, consider turning off the "verbose" option or running the code directly in the command line (which has no delay).

For processing the output files, see some suggestions in section 1.3, where the topic is covered. It is also important to understand the assumptions of the model and what is being calculated, which you find in chapter 2.

1.2.2 3D Example: csv Input

In this tutorial, we will cover another type of input: the `.csv` file. By convention, the columns in a `.csv` file represent the x, y, and z coordinates of all the solid particles in the domain. Although this type of input is not the most memory efficient in terms of memory storage, it is easy and efficient to read from a programming perspective and is easy to use as an output for either Paraview or Python scripts.

Open the ED-cPSD application, and open the "3D" tab. Change the input type from "Stack `.jpg`" to ".csv". The input file provided provided in the "Example 3D csv" folder is named "rec_729_300_int1.csv", which is a reconstruction of a carbon electrode featured in Adam 2022 [2]. This structure contains 300 voxels in each direction, a total of $2.7 \cdot 10^7$ voxels. The porosity is approximately 72.9%.

For this type of input, the code cannot automatically determine the size, so the user has to enter the dimensions in terms of pixels. This example sets the height, width, and depth to 300. You can also select all of the possible outputs and provide names for the expected output files.

The last three options are the same as in the 2D case. This time, increase the number of threads to 8 to demonstrate the parallel computing capabilities of ED-cPSD. Click "Generate" to see the a preview

of the input file, which is displayed in the right-hand side panel on the screen. The window should look something like figure 1.3.

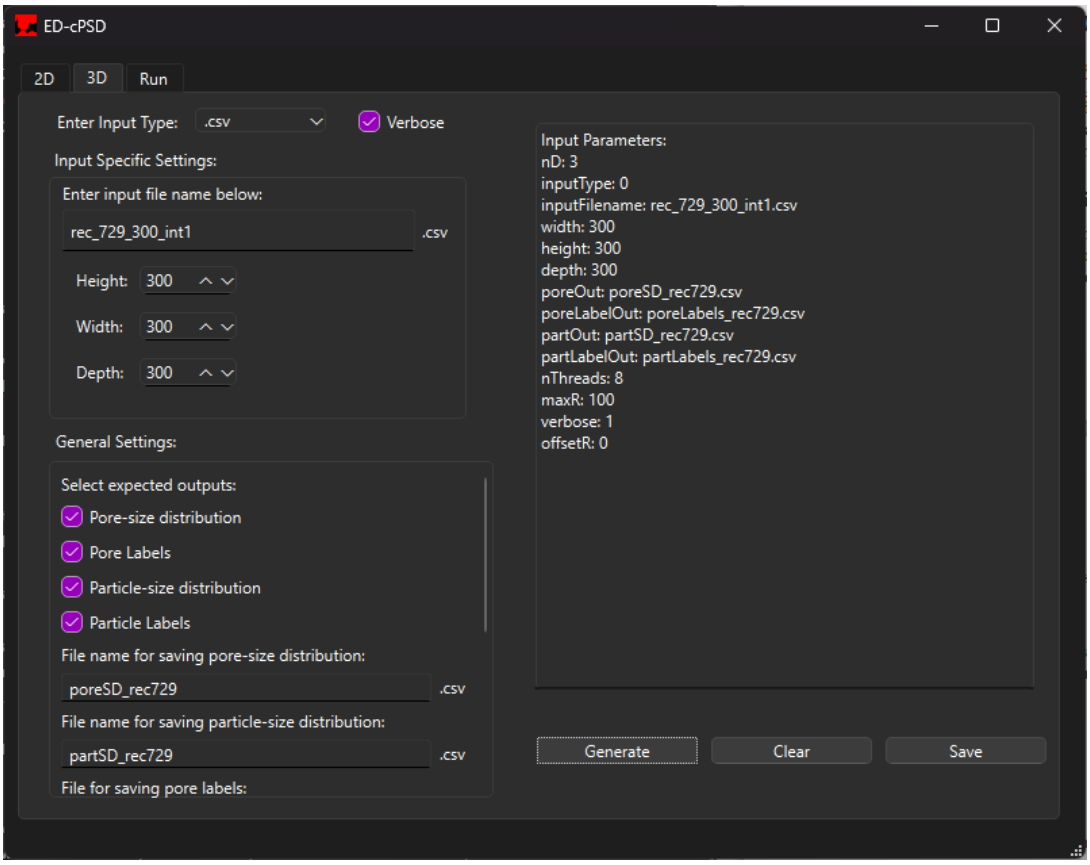

Figure 1.3: Example input file on ED-cPSD for the 3D example.

Save the input file in the same folder as the 3D structure `.csv` file. Then, switch over to the "Run" tab on the ED-cPSD software, and select the same folder again. Press the "Run" button and wait for the simulation to complete. You should see the outputs being produced and saved in the same folder as the original structure and the input file. For details of output processing, refer to section 1.3.

1.2.3 3D Example: Stack of .jpg

For this example, I will not provide a file. Instead, let's use a file from an open-source dataset as the example. Navigate to the NREL Battery Microstructure Library `https://www.nrel.gov/transportation/microstructure.html`, select the option to download samples. While there are several download options, I will use the segmented option for NMC-1-CAL where "1" denotes the active material, and "0" represents everything else.

Once the `.tif` file is downloaded, open it using ImageJ. The software will correctly interpret the file as a sequence of 2D images. Our task is to segment it properly and save the inputs. While the image is already binary, the lowest bit depth in most conventional file formats is 8-bit. Thus, we will use ImageJ to further segment the image from $0 \rightarrow 1$ to $0 \rightarrow 255$. To do this, go to "Image", then "Adjust", and then "Threshold". Make sure this is applied to all images in the stack.

Create a folder where you want to save the image slices. In ImageJ, navigate to "File", then "Save As", and select "Image Sequence". This will open a new dialog box with several options. Choose the folder you created, select the image format as "JPEG", and clear the image name field. The default options for "Star at" and "Digits" should be sufficient. Check the box "use slice labels as file names". The window should look like figure 1.4.

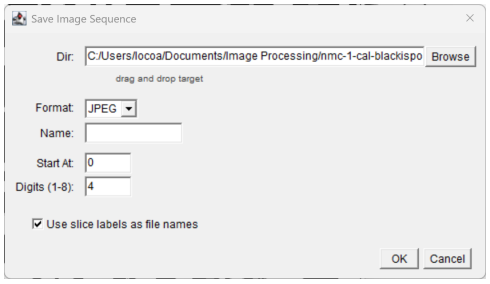

Figure 1.4: Screenshot of what the "Save Image Sequence" should look like on ImageJ side.

After saving, make sure that under "Image" and "Type" it says "8-bit". If some other option is selected, change it to 8-bit and re-save the stack as `.jpg`. If all of the aforementioned steps were followed, you should have a folder containing a stack of `.jpg` files. The files should have 4 digits with leading zeroes, with the first file named "0000.jpg". Now, in the ED-cPSD software, under the 3D tab, select the input type "Stack .jpg". Enter the stack size, which is the number of images in the stack (320 in this case), and the number of leading zeroes is the same as "Digits" from ImageJ, figure 1.4. The other options are still the same from the 2D and 3D case.

Press "Generate" and save the input file in the same folder as the figures. Now, go to the "Run" tab, select the folder again, and press the "Run" button to start the simulation.

1.3 Processing Output

The phase-size distributions are interpreted consistently in both 2D and 3D, as outlined in subsection 1.3.1. The subsection provides information on transforming the output into a size distribution based on diameter, include pixel resolutions, and obtaining the average diameter, D_{50}. However, plotting the particle labels by radius can be more complex in 3D. Therefore, the discussion is divided into a 2D discussion in subsection 1.3.2 and a 3D discussion in subsection 1.3.3.

1.3.1 Phase-Size Distribution Output

Regardless if the data was originally in 2D or in 3D, the format of this output remains the same. The first column contains the pore radius, labeled "r", and the second column contains the probability density of a given particle size "r", labeled "p(r)". The first column can easily be converted to the real diameter using equation 1.1.

$$d = 2 \cdot r \cdot pr \tag{1.1}$$

where pr is the pixel resolution of the base image.

An average diameter can be obtained from that in two ways. The simplest way to obtain D_{50} is by equation 1.2

$$D_{50} = \sum_{d_{min}}^{d_{max}} d \cdot p\,(d) \tag{1.2}$$

Another way to obtain D_{50} is to by calculating a cumulative probability distribution, as demonstrated in other articles [3, 4]. This method involves interpolating to determine the particle diameter at which the cumulative probability distribution reaches 0.5. Both methods yield the same D_{50}.

1.3.2 Phase-Labels in 2D

The ED-cPSD label files have 4 columns in 2D: the x and y pixel coordinates, the radius label "R", and the label "L". For more information on how "R" and "L" are derived, see section 2.4. This brief tutorial will show how to plot a colormap of "R" on the original image domain. The output can be very informative for visualizing the approximate locations and concentrations of certain particle sizes and shapes. This also provides a good understanding of how the the ED-cPSD code measures objects.

To plot the colormap, any plotting algorithm can be used. For this tutorial, we will use Python with the libraries `matplotlib`, `pandas`, and `numpy`. The file used in this tutorial is included in the GitHub

folder, under `/Examples/Contour Plot 2D`, with all the necessary files also included. The code below is well commented and straight forward.

The code below produces figure 1.5 using the data from subsection 1.2.1.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

plt.rcParams["font.family"] = "Times New Roman"

# image size properties
xSize = 2000
ySize = 2000
pixel_Resolution = 1  # units

# Read image
imgName = "cRec2000.jpg"
img = np.uint8(mpimg.imread(imgName))

# Read .csv
df1 = pd.read_csv("def_partLabelOut.csv")

# get raw data from csv files
raw_data = df1[["x", "y", "R", "L"]].to_numpy()
R = np.zeros((ySize,xSize))

x = raw_data[:,0]
y = raw_data[:,1]

for i in range(len(raw_data[:,0])):
 R[y[i]][x[i]] = raw_data[i,2]*pixel_Resolution

# Create the mesh grid
Xp, Yp = np.meshgrid(np.linspace(0, 1, xSize), np.linspace(1.0*ySize/xSize, 0, ySize))

# plotting
fig1, ((ax1, ax2)) = plt.subplots(1, 2, constrained_layout=True)
fig1.set_dpi(100)
fig1.set_size_inches(8, 4)

# First axis is just the image
ax1.imshow(img)
ax1.set_title(imgName, fontsize=16)

# Second axis is R - contour
CS2 = ax2.contourf(Xp, Yp, R, 40, cmap=plt.cm.rainbow)
cbar2 = fig1.colorbar(CS2, ax=ax2, fraction=0.046, pad=0.04)
cbar2.set_label(r'Particle Radius [voxels]', rotation=90, fontsize=14)
ax2.set_title("Particle Radius Distribution", fontsize=16)
ax2.set_aspect('equal', adjustable='box')
plt.show()
```

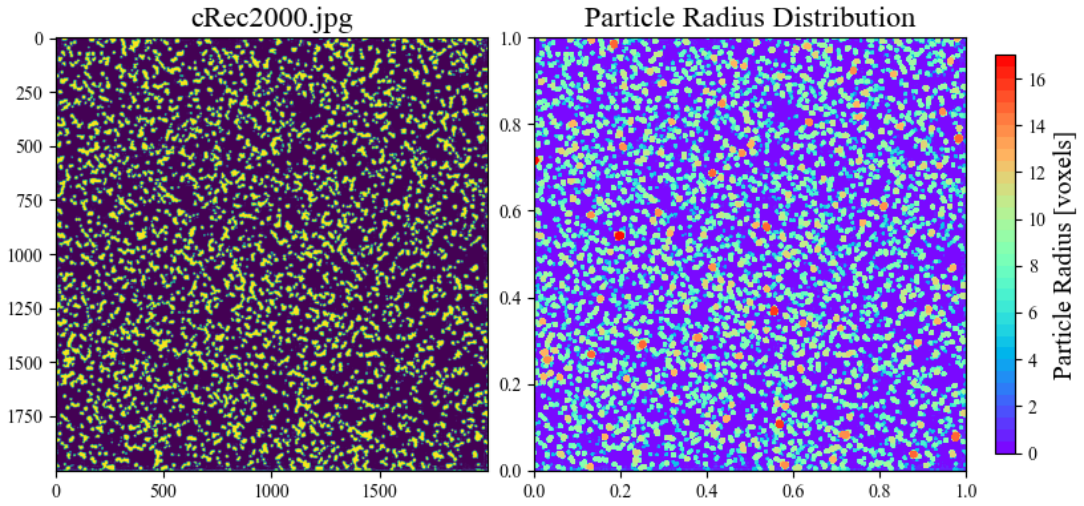

Figure 1.5: Example contour plot from processed ED-cPSD output.

1.3.3 Phase Labels in 3D

In this brief tutorial, I will show another way of displaying the phase labels similar to subsection 1.3.2. While this can still be done using Python, this tutorial will instead showcase a technique using the open-source software Paraview [5].

The ED-cPSD label files in 3D have 5 columns: the x, y, and z pixel coordinates, the radius label "R", and the label "L". The output `.csv` file can be read directly into Paraview. Simply open the software, select "File", then "Open", and import the particle labels output from example 1.2.2. Make sure to select the "Global CSV Reader". A menu will appear on the left-hand side of the screen. Click "Apply" to accept the default information. Once the software finishes loading the data, the "SpreadSheetViewer" will appear on the screen but can be closed immediately. Select the loaded file on the pipeline browser on the left-hand side of the screen, then select the "Filters" tab at the very top of the page, search for "TableToPoints" filter, and select it.

New options will appear on the left-hand side menu. Under the column, select "x" for the X column, "y" for the Y column, and "z" for the Z column. Click "Apply". This will generate a rendering of the 3D structure, but it may still be difficult to identify features or read contours, thus several modifications are needed at this step. On the left-hand side menu, click on the cog icon next to the search bar to "toggle advanced options". Make sure those options are available. Under "Representation", change from "Surface" to "3D Glyphs". Below this, in the "Coloring" section, change the option from "solid" to "R". Paraview will automatically use the radius labels "R" to apply a colormap to the surface. Scroll down to "Glyph Parameters", under "Scale Array", make sure "None" is selected. Change the "Glyph Type" from "Arrow" to "Box". At this stage, a good visualization is already obtained.

Optional steps to further improve the output quality include changing the background to a solid color with better contrast, adjusting the colorbar settings, and modifying the axis and axis labels. By following these steps (including the optional ones), we arrive at figure 1.6.

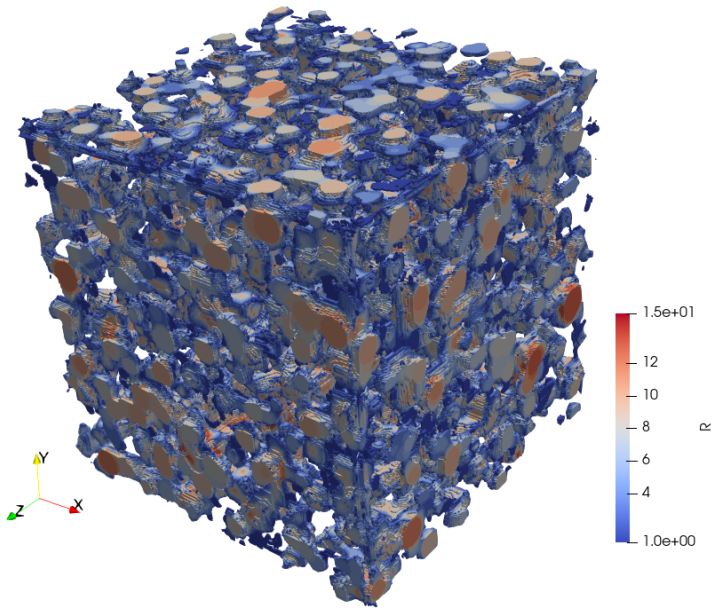

Figure 1.6: Example contour plot from processed ED-cPSD output for a 3D structure.

1.4 Command Line Operation

As explained in chapter 2, the code offers flexibility to run both in the GUI and as a standalone command line application. The command line mode is significantly more efficient and allows researchers to utilize the algorithm in HPC environments, where a GUI is often not available. This can be easily accomplished with this project. The source files that required for command line operation are available on GitHub in the "src" folder. Simply add those files to your project folder and compile the code as follows (assuming a Linux environment):

```
g++ -fopenmp ./ED_PSD_CPU.cpp -o ./ExecutableName.out
```

This will create an executable named `ExecutableName.out`. This step only needs to be done once, as all simulation parameters can subsequently be controlled via `input.txt`. Note that the input file can still be generated using the GUI and then transferred to the HPC environment. Additionally, modifying the input text file with a bash script is the easiest way to run large batches of simulations automatically.

Chapter 2

Algorithm Details

2.1 Motivation

The interplay between different phases in heterogeneous materials [6] is of paramount importance in determining the bulk-scale properties of said materials. In porous media, more specifically, the interplay between the particle and pore spaces is critical for determining the transport properties, as thoroughly discussed in various studies[7, 8, 9, 10, 11, 12, 13].

Several software packages and algorithms aim to reconstruct, analyze, and characterize the morphology of such complex systems, including MCRpy [14] and MATBOX [15]. However, there are varying interpretations of the meaning and accuracy of physical descriptors. One such property is the particle/pore/phase size distribution, often reported as the average pore/particle diameter, D_{50}. This is problematic because there is frequently no clear definition of what constitutes a particle or a pore, let alone a consistent determination of what "size" means without context (e.g. the size of an ellipsoid could refer to the semi-major axis, the semi-minor axis, the average diameter, the length, or an average of all of those).

In some materials, like carbon-based materials for battery electrodes or certain types of sandstone, scanning electron microscopy (SEM) reveals that they are largely composed of small, approximately spherical individual particles. In these cases, segmenting particles within the domain and assigning them sizes is feasible, which is the approach of discrete particle-size distribution (dPSD) algorithm.

While different dPSD algorithms may employ various methods, the basic scheme remains the same: each domain is segmented into particles, and each particle is labeled separately. The morphology of each particle is then quantified, and averages, such as PSD and D_{50}, can be obtained. More details and a brief compilation of such algorithms can be found in Usseglio-Viretta (2020) [4].

Another avenue for obtaining the PSD is via continuous pore-size distribution (cPSD) algorithms. This class of algorithms continuously assess the size of each phase space, without the need for segmentation. This interpretation is more consistent for materials like stochastic aluminum foams [16], where both the aluminum and pore space are continuous and amorphous, making dPSD algorithms incompatible. Even in better structured materials like triply-periodic minimal surfaces (TPMS), the cPSD measurements of particle and pore size distributions are physically interpretable, representing the probability density of thickness distribution and pore-space minimum radius distribution, respectively. In the aforementioned examples of carbon-based battery electrodes and sandstone, the pore-space is simply the interstitial space defined by the absence of solid particles, making it continuous and amorphous. Using a dPSD algorithm in such cases can lead to over-segmentation [4]. Furthermore, the cPSD class of algorithms more closely resemble experimental methods for determining pore-size distributions, such as mercury intrusion porosimetry [3] or nitrogen desorption porosimetry [17]. A brief review and further discussion can be found in Usseglio-Viretta (2020) [4].

However, the cPSD algorithms have several downsides [4], including the need for data fitting, strong assumptions about particle shape (the spherical-particle assumption), lack of information about the location and anisotropy of the related spaces, no method for particle identification, requirement for large representative volumes for accurate assessment, and often underestimation of average phase sizes.

2.2 cPSD via Sequential Erosion-Dilation: ED-cPSD

From any base 2D or 3D digital structure, the domain can be represented as a binary boolean array, b. In b, a true entry represents the phase of interest, and false represents everything else. In the case of pore-size distribution, the pore space is considered true, while the particle space is false. For particle-size distribution, the pore space is false, and the particle space is true. This algorithm is also flexible enough to measure the phase-size distribution of multiple solid phases, where the phase of interest is designated as true, and everything else as false.

The algorithm begins by expanding the false regions of the array b. Initially, at a radius $r = 1$, every true entry within a radius r of an interface is converted to false. The results are stored in array D. In the second stage, the algorithm shrinks every false region within a radius r of a true entry in array D, and the result is stored in array E. At the end of each iteration, the total number of true entries in arrays b, D, and E are stored, and r is increased by one.

It is evident from this basic algorithm that any region in b that has a diameter less than $2r$ will cease to exist in D. If a region is not completely erased at a radius r in the transformation from b to D, then it will reappear after the operation from D to E in the exact same shape as it was in b. At each radius r, there is a new E_r, starting with $E_0 = b$. By comparing E_r to E_{r-1}, all the regions that are removed from E at r, but not at $r-1$ can be identified. Equation (2.1) below describes how a cPSD distribution, $\Phi(r)$, can be obtained from the combination of E_r, E_{r-1} and b.

$$\Phi(r) = \frac{\sum(E_r - E_{r-1})}{\sum b} \tag{2.1}$$

From an implementation perspective, it suffices to store a single integer array $e_r = \sum E_r$, instead of storing E_r for each r. The distribution is a probability distribution, ensuring that $\sum_{r=1}^{r_{\max}} \Phi(r) = 1$, where $r_{\max}$ is the radius at which $\sum E_{r_{\max}} = 0$. This algorithm is straightforward to interpret both visually and physically, as illustrated in figure 2.1.

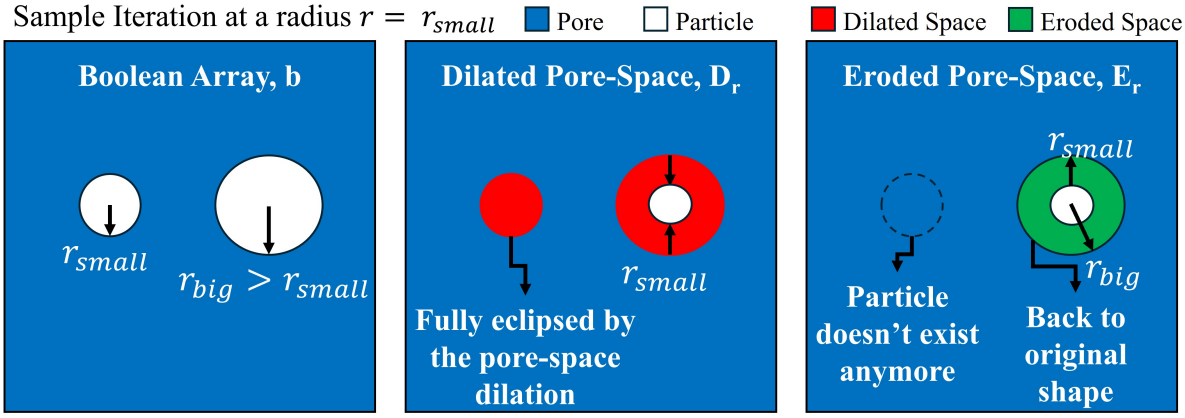

Figure 2.1: Example schematic representing the logic behind the computation of particle size distribution for a domain containing two spherical particles of different radii.

2.3 Erosion-Dilation Implementation

In a previous study [2], the exact algorithm described in Section 2.2 was employed to calculate the PSD of large 3D structures consisting of millions of voxels. In that study, however, the erosion and dilation operations were calculated naively from each pixel. Specifically, at each iteration, the interface between true and false values across the entire b array was mapped, and for each pixel at the interface, a radius r must be scanned to perform the dilation. This process was then repeated for D to generate E. Consequently, at each radius r, the algorithm exhibited a computational complexity of $O\left(Nr^3\right)$, where N is the total number of voxels in the domain. This approach becomes problematic in domains with large pores or particles, as the algorithm scales poorly with increasing r.

In this study, we propose an alternative by calculating the exact Euclidean distance transform (EDT), and using the EDT to perform erosion and dilation operations. Evidence [18] suggests that the fastest method to calculate the exact EDT is through the Meijster algorithm [19], which is also known as the

Felzenszwalb algorithm [20]. Both authors arrived at an equivalent solution independently, but since Meijster was published earlier, we will refer to the algorithm as Meijster's algorithm.

The Meijster algorithm calculates the EDT using four linear passes in 2D and six linear passes in 3D, therefore the algorithm is classified as linear in complexity, $O(N)$. As the studies related to porous media can involve domains with trillions of voxels [21], employing an algorithm with linear time complexity is quintessential, and represents a significant improvement over the naive approach [2].

Both Meijster [19] and Felzenszwalb [20] claim that the algorithm is scalable to an arbitrary number of dimensions, although neither provides a detailed description of the process for calculating EDT in 3D. In 2D, the EDT can be calculated as follows:

$$\mathrm{EDT2D}(x,y) = \min\left(i : 0 \leq i < m : (x-i)^2 + G(i,y)^2\right) \tag{2.2}$$

where $G(i,y)^2 = \min(j : 0 \leq j < n \wedge b[i,j] : |y-j|)$. In 2D, the Meijster algorithm has two phases. In the first phase, $G(x,y)$ is calculated for each column (fixed $x = i$). In the second phase, the EDT2D is calculated for each row, y, according to $G(i,y)$ and equation (2.2). To adapt the algorithm for use in 3D, only one additional phase is necessary. In the first phase, $G(x,y,z)$ is calculated in the exact same manner, for each column in the entire domain (fixed x and z). Phase 2 also remains the same, in which EDT2D is calculated for each row (fixed y and z). Phase 3 repeats phase 2, with the only change being that EDT3D depends on EDT2D, not $G(x,y,z)$. For each slice (fixed x and y), EDT3D is calculated according to equation (2.3).

$$\mathrm{EDT3D}(x,y,z) = \min\left(k : 0 \leq k < p : (z-k)^2 + EDT2D(x,y,k)\right) \tag{2.3}$$

Beyond reducing the time complexity of the algorithm, calculating the EDT before erosion or dilation has an additional benefit. In the algorithm presented in Section 2.2, an EDT in the true phase is necessary to transform from b to D_r. The transformation follows the simple rule that if $\mathrm{EDT3D}(x,y,z)\,|_{b=1} \leq r^2$, then $D(x,y,z)_r = 0$. Whether $\mathrm{EDT3D}|_{b=1}$ is calculated from b or E_{r-1}, the dilation yields the same D_r. Therefore, $\mathrm{EDT3D}|_{b=1}$ only needs to be calculated once in the entire algorithm.

The EDT for the false phase of D is necessary to transform D_r into E_r at each iterative step. In other words, for each r, one EDT3D must be calculated using the Meijster algorithm in 3D. Since this is by far the most time consuming operation, each iteration of the successive dilation-erosion algorithm takes approximately the same time it takes to create one EDT map.

Finally, each phase of the Meijster algorithm can be thought of as having a set of independent operations, making it an embarrassingly parallel algorithm [22]. That means that on a cluster with k threads (independent of being on CPU or GPU), the time complexity of generating the EDT should be at or near $O\left(\frac{N}{k}\right)$. This type of algorithm scales extremely well with additional computational resources in HPC environments.

2.4 Phase Labeling

Most cPSD algorithms do not have any mechanisms to backtrack a phase segmentation from the calculation of phase-size distribution [4]. This can be considered as one of the downsides of cPSD algorithms, as there is no mechanism to retrieve particle morphology or inhomogeneity from the cPSD metric.

This is not the case for the ED-cPSD. In fact, the base software produces two different segmentation metrics. One is displayed in sections 1.3.2 and 1.3.3, and that is the labeling via radius of removal.

This labeling mechanism based on radius emerges directly from the algorithm discussed in section 2.2. An array R is initialized at the beginning of the phase-size distribution, and it is all initialized to zeroes. Then, at any given radius, r, the array E_r is compared to the original boolean array, b. If $b|_{i,j,k} == 1$, $E_r|_{i,j,k} == 0$, and $R|_{i,j,k} == -1$, the $R|_{i,j,k} = r$.

In simpler terms, array R keeps track of which r eliminated pixel coordinates (i,j,k). In order to further group pixels by radius of removal, another step is necessary to label clusters of pixels with the same radius at the same location.

A flood-fill algorithm `https://en.wikipedia.org/wiki/Flood_fill` is applied at the pixel level for this task, grouping clustered pixels with the same r. This is how labels L are derived. This process is done in ascending order of r. If it were done in descending order of r, it would be analogous to a Watershed algorithm [23, 21], albeit at a lower resolution and without segmentation post-processing.

2.5 GUI and the Qt Framework

The Qt framework `https://www.qt.io/` is used to wrap the C++ code into a GUI. The publication outlines a flexible double computation model that allows the same functions to be called from inside the GUI or directly from the command line.

The GUI serves a dual purpose: it aids in the creation of input files and it can also be used to run the code. Creating input files is relatively straight forward and involves connecting the buttons to the desired outputs using the signals and slots built in to Qt. Therefore this process is not covered in this document. However, running the code within the GUI is more complex, and deserves detailed attention. Understanding the intricate interplay between the C++ execution, the Qt framework, the GUI execution loop, the worker thread responsible for the simulation, and the additional threads recruited by OpenMP for parallel execution is fundamental for grasping certain decisions regarding the code structures. The main challenge in running a resource intensive simulation as part of a GUI is that the GUI execution loop must constantly run. Therefore, the simulation cannot be executed within the same execution loop, as that will freeze the GUI and cause the program to crash.

Several methods were considered to avoid this issue. Without going into detail about the methods that were unsuccessful, the goal was to find a balance between independent execution and maintaining communicate between the GUI execution loop and the code execution. Methods like Qt::Concurrent allow for the independent execution, but lack communication. The method used involves separating the GUI execution loop by defining a class for the main window, and a class for the worker thread. The worker thread has an execution loop that is entirely independent of the execution loop of the main window. Communication between the two occurs via shared variables locked by a mutex, and the worker can send messages to the main window via signals. This computational model is best illustrated by the Mandelbrot example from the Qt documentation, which can be found here: `https://doc.qt.io/qt-6/qtcore-threads-mandelbrot-example.html`.

2.6 Modifying Base Algorithm

This section refers to the `ED_cPSD_CPU.cpp` and `.hpp` files included in the `src` folder on GitHub. These files are only used for command line operation. The files used for the GUI operation are located in a different folder, as covered in section 2.7.

The structure of the code is fairly simple: the main `.cpp` file is responsible for reading the input file, storing the information in the appropriate data structures, and deciding between 2D and 3D operation based on the user input. The `.hpp` file contains everything else. The code is not lengthy, hence it was more convenient to have all functions and data structures in one file. There are three essential data structures. The struct *options* handles all user-entered options. The structs *sizeInfo* and *sizeInfo2D* handle mesh parameters for 3D and 2D simulation modes, respectively.

The functions that actually control the simulations are called `Sim2D` and `Sim3D`, both of which take a pointer to *options* as input. Structurally, these functions are very similar: they read the input files, declare necessary structure arrays, and call other functions to run pore and/or particle size distributions according to the user input. Each function has a header explaining its expected inputs and outputs, as well as describing its functionality. Overall, the code is concise and well-commented, making it easy to modify. Below is an example of a function declaration and comment header:

```
int partSD_2D(options *opts,
              sizeInfo2D *info,
              char *P,
              char POI)
{
    /*
        Function partSD_2D:
        Inputs:
            - pointer to options struct
            - pointer to structure info struct
            - pointer to phase-array
            - char phase of interest
        Outputs:
            - None.
```

```
    Function will calculate particle size distribution of array P.
*/

...
```

2.7 Modifying GUI Code

The GUI code follows many of the same conventions already covered in section 2.6. However, the structure of the files is quite different, as there are many more files involved in this mode of operation. Additionally, several functions are re-formatted into a class structure. Although they behave the same, the implementation is slightly different.

This version of the code includes significantly more files than the simple command line version. The project is structured around the `CMakeLists.txt` file, which allows Qt and CMAKE to organize all the files into a cohesive executable. The structure of the `CMakeLists.txt` file won't be covered. There is one singular file with a `.ui` extension. This is the Qt user interface file, and can be opened via the Qt Designer or via the Qt Creator "Design" mode. This file contains all information about the layout of the GUI.

This project includes two main source files: `main.cpp` and `mainwindow.cpp`. The first file is responsible for starting the QApplication and initiating the execution loop of the main window. The second file contains all of the functions, slots, and signal connections, as well as class declarations. To understand those, we must first understand the helper files structure.

There are four helper files in total.

1. `data_structs.hpp`: This file contains the data structure definitions, identical to those in the command line version. This separation allows the classes to inherit or use these data structures as input for protected functions. Leaving the data structures definition in the `ED_PSD_CPU.hpp` would create issues with recursive definitions.

2. `stb_image.h`. This file, from `https://github.com/nothings/stb`, is used for reading the `.jpg` files.

3. `main_window.h`. This contains the class definitions for the two classes used in this work. The *MainWindow* class handles all information and adds functionality for the main window's buttons. This class is the backbone of the GUI, and is also used to invoke the *Worker* class. This class controls the simulation and re-implements the QThread class. This runs mostly in the background and handles the communication between the two classes via signals. A mutex is also common to both *Worker* and *MainWindow*, and is implemented so that *MainWindow* can pass essential information, such as orders to abort the simulation or close the program, to *Worker*.

4. Auxiliary functions, such as reading images or calculating EDT, remain in `ED_PSD_CPU.hpp` since they don't require much communication or are executed quickly. Hence they don't need to be part of the *Worker* class.

The function headers follow a similar convention to the one defined in section 1.4, making the code readable and easy to modify. If there are any further questions about the implementation, don't hesitate to reach one to the corresponding author.

Finally, the `resources` folder contains a `.ico` file, which is used as the icon for the application. If all of these files are present in the computer, and Qt is added to the path, the project should be easy to build with CMAKE. However, if Qt is not on the path, either use the static version of the project available on GitHub with manually linked libraries, or consider using the Qt Creator (Qt's own IDE) to build the project. For new users, installing Qt and using the Qt Creator is highly recommended.

Chapter 3

Case Studies

This section will showcase examples of work involving the ED-cPSD software. As more studies are published, this section will grow accordingly. We encourage users to share any results that want to be featured here, this is a great way to increase the visibility to your work!

3.1 Case Study 1: NREL Battery Microstructure Dataset

This case study is included for validation purposes and showcases the particle-size distribution capabilities of this novel algorithm. Seven 3D electrode structures from the National Renewable Energy Lab (NREL) battery microstructure dataset were used [24, 25], and the particle-size distributions generated by the ED-cPSD software are shown in figure 3.1.

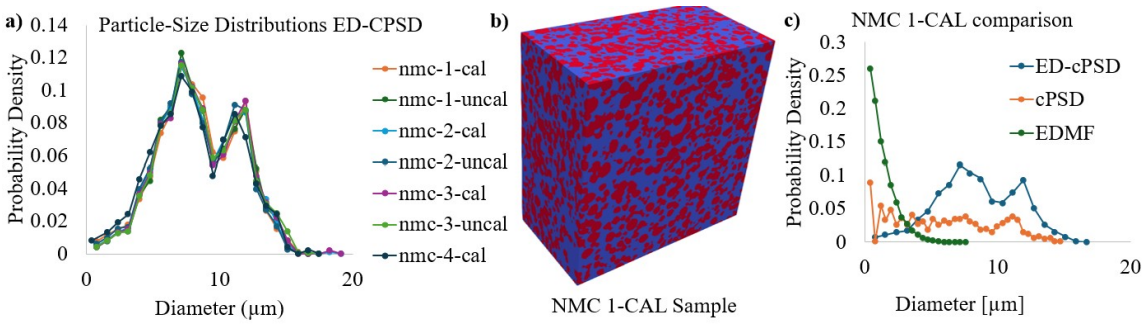

Figure 3.1: **a)**Particle-size probability density of all seven NMCs from NREL's battery microstructure dataset as generated by ED-cPSD. **b)** Visualization of the NMC 1-CAL sample. **c)** Comparison of ED-cPSD with the other continuous phase-size distribution algorithms available in MATBOX.

The probability density, $\Phi(r)$, is used to calculate the average particle diameter, D_{50}, as shown in equation 3.1.

$$D_{50} = \sum_{r=1}^{r_{max}} \Phi(r) \cdot (2r) \tag{3.1}$$

Applying equation 3.1 to the data in figure 3.1**a)**, the samples yield D_{50} values ranging from $8.23\mu m$ to $8.61\mu m$. The ED-cPSD offers a more granular and realistic D_{50} measure compared to the continuous PSD [3, 26], which reports [4] D_{50} between $5.5\mu m$ and $6.2\mu m$ for the same data and is known to severely under-estimate D_{50} [4]. Figure 3.1**c)** shows a comparison in size distribution between the two continuous methods available in MATBOX and the ED-cPSD method proposed in this study. It is evident that EDMF (based on the size distribution generated, not the data fitting) and cPSD methods severely underestimate particle sizes and don't provide smooth distributions. The ED-cPSD method does not require any data fitting and D_{50} is a direct output from the probability distributions and equation 3.1.

In comparison, the D_{50} from dPSD methods reported [4] a range from roughly $10\mu m$ to $11\mu m$ for the same data. The dPSD methods in Usseglio-Viretta 2020 [4] reportedly took between one to two weeks to evaluate a single representative element from a 3D version. A more recent version of the PCRF [27]

algorithm using instance segmentation by parts has a wall-time of one to two days for half a billion voxels with 10 CPU cores. This is between 240 and 480 CPU-hours for half a billion voxels. For a rough comparison, the ED-cPSD algorithm takes around two minutes with 8 CPU cores (0.25 core-hours) for 130 million voxels for pore and particle size distributions of the structure in figure 3.1**b)**.

Note that for the discrete methods, the domain is segmented and each particle is then characterized. In contrast, the ED-cPSD method provides a continuous statistical assessment of the particle space without segmentation. This discrepancy in continuous versus discrete modeling is shown in figure 3.2.

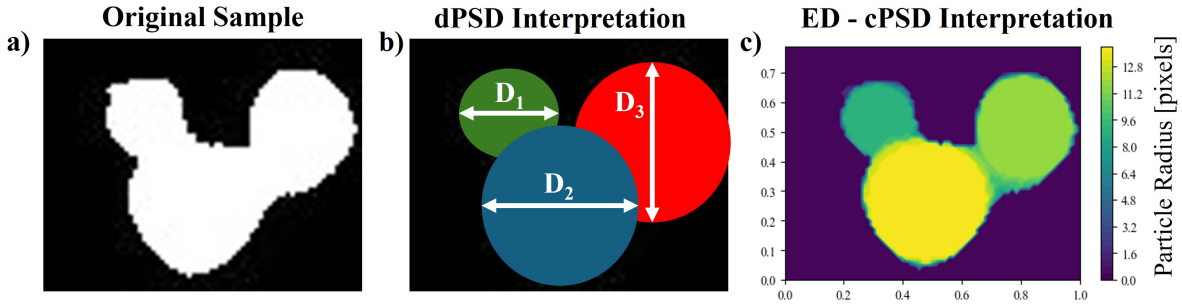

Figure 3.2: **a)** shows a single particle extracted from the structure shown in figure 3.1. **b)** Depiction of discrete domain segmentation manually extracted from an EDT. **d)** Colormap of the particle-space labels by extracted radius, output from the ED-cPSD software.

From figures 3.2**b)** and **c)**, the difference between the two approaches becomes more obvious: the ED-cPSD algorithm measures particle-space diameters continuously, without any segmentation, while the dPSD algorithms will segment, and then measure. Thus, to obtain a smooth $\Phi(r)$ using a dPSD algorithm, significantly more particles are needed in the domain.

Chapter 4

Additional Information

4.1 Acknowledgments

This work used Expanse(GPU)/Bridges2(CPU) at SDSC/PSC through allocations MAT210014 and MAT230071 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296 [28].

X.L. highly appreciates the support from the National Science Foundation (Award 1941083 and 2329821).

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number 18/CJ000/08/08.

4.2 License

Bibliography

[1] Schneider, C. A., Rasband, W. S. & Eliceiri, K. W. Nih image to imagej: 25 years of image analysis. *Nature Methods* **9**, 671–675 (2012). URL http://dx.doi.org/10.1038/nmeth.2089.

[2] Adam, A., Wang, F. & Li, X. Efficient reconstruction and validation of heterogeneous microstructures for energy applications. *International Journal of Energy Research* (2022). URL https://onlinelibrary.wiley.com/doi/full/10.1002/er.8578.

[3] Münch, B. & Holzer, L. Contradicting geometrical concepts in pore size analysis attained with electron microscopy and mercury intrusion. *Journal of the American Ceramic Society* **91**, 4059–4067 (2008).

[4] Usseglio-Viretta, F. L. E. *et al.* Quantitative relationships between pore tortuosity, pore topology, and solid particle morphology using a novel discrete particle size algorithm. *Journal of The Electrochemical Society* **167**, 100513 (2020). URL https://iopscience.iop.org/article/10.1149/1945-7111/ab913bhttps://iopscience.iop.org/article/10.1149/1945-7111/ab913b/meta.

[5] Ahrens, J. P., Geveci, B. & Law, C. C. Paraview: An end-user tool for large-data visualization. In *The Visualization Handbook* (2005). URL https://api.semanticscholar.org/CorpusID:56558637.

[6] Wu, X. & Zhu, Y. Heterogeneous materials: a new class of materials with unprecedented mechanical properties. *Materials Research Letters* **5**, 527–532 (2017).

[7] Lin, G. *et al.* Effect of pore size distribution in the gas diffusion layer adjusted by composite carbon black on fuel cell performance. *International Journal of Energy Research* **45**, 7689–7702 (2021). URL https://onlinelibrary.wiley.com/doi/full/10.1002/er.6350https://onlinelibrary.wiley.com/doi/abs/10.1002/er.6350https://onlinelibrary.wiley.com/doi/10.1002/er.6350.

[8] Cui, C. L., Schweich, D. & Villermaux, J. Influence of pore diameter distribution on the determination of effective diffusivity in porous particles. *Chemical Engineering and Processing* **26**, 121–126 (1989).

[9] Liu, X. *et al.* The influence of pore size distribution on thermal conductivity, permeability, and phase change behavior of hierarchical porous materials. *Science China Technological Sciences* **64**, 2485–2494 (2021). URL http://dx.doi.org/10.1007/s11431-021-1813-0.

[10] Adam, A., Fang, H. & Li, X. Effective thermal conductivity estimation using a convolutional neural network and its application in topology optimization. *Energy and AI* **15**, 100310 (2024). URL http://dx.doi.org/10.1016/j.egyai.2023.100310.

[11] Li, J. X., Rezaee, R., Müller, T. M. & Sarmadivaleh, M. Pore Size Distribution Controls Dynamic Permeability. *Geophysical Research Letters* **48**, e2020GL090558 (2021). URL https://onlinelibrary.wiley.com/doi/full/10.1029/2020GL090558https://onlinelibrary.wiley.com/doi/abs/10.1029/2020GL090558https://agupubs.onlinelibrary.wiley.com/doi/10.1029/2020GL090558.

[12] Tian, S., Ren, W., Li, G., Yang, R. & Wang, T. A Theoretical Analysis of Pore Size Distribution Effects on Shale Apparent Permeability. *Geofluids* (2017). URL https://doi.org/10.1155/2017/7492328.

[13] Kapat, K. *et al.* Influence of Porosity and Pore-Size Distribution in Ti 6 Al 4 V Foam on Physicomechanical Properties, Osteogenesis, and Quantitative Validation of Bone Ingrowth by Micro-Computed Tomography. *ACS Appl. Mater. Interfaces* (2017). URL www.acsami.org.

[14] Seibert, P., Raßloff, A., Kalina, K., Ambati, M. & Kästner, M. Microstructure characterization and reconstruction in python: Mcrpy. *Integrating Materials and Manufacturing Innovation* **11**, 450–466 (2022). URL https://doi.org/10.1007/s40192-022-00273-4.

[15] Usseglio-Viretta, F. L. *et al.* Matbox: An open-source microstructure analysis toolbox for microstructure generation, segmentation, characterization, visualization, correlation, and meshing. *SoftwareX* **17** (2022).

[16] Stallard, S., Jiang, H., Chen, Y., Bergman, T. L. & Li, X. Exploring the design space of the effective thermal conductivity , permeability , and stiffness of high-porosity foams. *Materials & Design* **231**, 112027 (2023). URL https://doi.org/10.1016/j.matdes.2023.112027.

[17] Joyner, L. G., Barrett, E. P. & Skold, R. The determination of pore volume and area distributions in porous substances. ii. comparison between nitrogen isotherm and mercury porosimeter methods. *Journal of the American Chemical Society* **73**, 3155–3158 (1951).

[18] Fabbri, R., Da F. Costa, L., Torelli, J. C. & Bruno, O. M. 2D Euclidean distance transform algorithms: A comparative survey. *ACM Computing Surveys* **40** (2008).

[19] Meijster, A., Roerdink, J. B. T. M. & Hesselink, W. H. *A General Algorithm for Computing Distance Transforms in Linear Time*, 331–340 (Springer US, Boston, MA, 2000). URL https://doi.org/10.1007/0-306-47025-X_36.

[20] Felzenszwalb, P. F. & Huttenlocher, D. P. Distance transforms of sampled functions. *Cornell Computing and Information Science Technical Report TR20041963* **4**, 1–15 (2004). URL https://theoryofcomputing.org/articles/v008a019http://ecommons.cornell.edu/handle/1813/5663%5Cnhttp://ecommons.library.cornell.edu/handle/1813/5663.

[21] Barnes, R. Parallel priority-flood depression filling for trillion cell digital elevation models on desktops or clusters. *Computers and Geosciences* **96**, 56–68 (2016).

[22] Wilkinson, B. & Allen, M. C. *Parallel Programming: Techniques and applications using networked workstations and parallel computers* (Pearson/Prentice Hall, 2005), 2 edn.

[23] Barnes, R., Lehman, C. & Mulla, D. Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. *Computers and Geosciences* **62**, 117–127 (2014).

[24] Usseglio-Viretta, F. L. E. *et al.* Resolving the discrepancy in tortuosity factor estimation for li-ion battery electrodes through micro-macro modeling and experiment. *Journal of The Electrochemical Society* **165**, A3403–A3426 (2018). Co-Variance approach to PSD.

[25] Usseglio-Viretta, F. L. E. & Smith, K. Quantitative microstructure characterization of a nmc electrode. *ECS Meeting Abstracts* **MA2017-01**, 1122–1122 (2017). URL https://iopscience.iop.org/article/10.1149/07711.1095ecsthttps://iopscience.iop.org/article/10.1149/07711.1095ecst/meta.

[26] Holzer, L. *et al.* Microstructure degradation of cermet anodes for solid oxide fuel cells: Quantification of nickel grain growth in dry and in humid atmospheres. *Journal of Power Sources* **196**, 1279–1294 (2011).

[27] Popeil, M. *et al.* Heterogeneity of the dominant causes of performance loss in end-of-life cathodes and their consequences for direct recycling. *Advanced Energy Materials* 2405371 (2025). URL https://onlinelibrary.wiley.com/doi/full/10.1002/aenm.202405371https://onlinelibrary.wiley.com/doi/abs/10.1002/aenm.202405371https://advanced.onlinelibrary.wiley.com/doi/10.1002/aenm.202405371.

[28] Boerner, T. J., Deems, S., Furlani, T. R., Knuth, S. L. & Towns, J. Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support. In *Practice and Experience in Advanced Research Computing*, PEARC '23, 173–176 (ACM, 2023). URL http://dx.doi.org/10.1145/3569951.3597559.